The final publication is available at ACM via http://dx.doi.org/10.1145/3392717.3392762

Wavefront Parallelization of Recurrent Neural Networks on Multi-core Architectures

Robin Kumar Sharma
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
robinkeralaam@gmail.com

Marc Casas
Barcelona Supercomputing Center (BSC)
Barcelona, Spain
marc.casas@bsc.es

ABSTRACT

Recurrent neural networks (RNNs) are widely used for natural language processing, time-series prediction, or text analysis tasks. The internal structure of RNNs inference and training in terms of data or control dependencies across their fundamental numerical kernels complicate the exploitation of model parallelism, which is the reason why just data-parallelism has been traditionally applied to accelerate RNNs.

This paper presents W-Par (Wavefront-Parallelization), a comprehensive approach for RNNs inference and training on CPUs that relies on applying model parallelism into RNNs models. We use fine-grained pipeline parallelism in terms of wavefront computations to accelerate multi-layer RNNs running on multi-core CPUs. Wavefront computations have been widely applied in many scientific computing domains like stencil kernels or dynamic programming. W-Par divides RNNs workloads across different parallel tasks by defining input and output dependencies for each RNN cell. Our experiments considering different RNNs models demonstrate that W-Par achieves up to 6.6× speed-up for RNN models inference and training in comparison to current state-of-the-art implementations on modern multi-core CPU architectures. Importantly, W-Par maximizes performance on a wide range of scenarios, including different core counts or memory hierarchy configurations, without requiring any change at the source code level.

KEYWORDS

Deep Neural Network (DNN), Wavefront Parallelization, Recurrent Neural Networks (RNNs), Long-Short Term Memory (LSTM), Gated Recurrent Units (GRUs), OmpSs, CPU Task Parallelism

1 INTRODUCTION

Neural networks composed of multiple layers are called Deep Neural Networks (DNNs) [38]. DNNs are widely used to automatically carry out activities like classification or pattern detection of text, images, speech, motions, or any other data format. While feed-forward [14] and convolutional DNNs [39] have shown a very remarkable capacity for pattern detection and classification of image data sets, they do not have any internal dynamic state describing connections between past and future data, which is a fundamental feature to successfully carry out activities like automatic speech recognition (ASR), speech translation (ST), and text-to-speech (TTS). Recurrent Neural Networks (RNNs) [51] contain memory units able to display dynamic and temporal connections between past and future data. The outstanding text and signal analysis properties of RNNs and other recurrent models like Long-Short Term Memories (LSTMs) [27] and Gated Recurrent Units (GRUs) [19] make them the prevalent choice to analyze sequential and unsegmented data like text or speech signals. The internal structure of RNN inference and training in terms of dependencies across their fundamental numerical kernels complicate the exploitation of model parallelism, which has not been fully exploited to accelerate forward and backward propagation of RNNs on multi-core CPUs.

This paper proposes W-Par, a parallel execution model for RNNs and its variants LSTMs and GRUs. W-Par exploits all possible concurrency available in forward and backward propagation routines of RNNs. These propagation routines display complex data and control dependencies that require sophisticated parallel approaches to extract all possible concurrency. W-Par represents RNNs forward and backward propagation as a computational graph [37] where nodes represent computation and edges identify data and control dependencies across them. A run-time system software takes responsibility for orchestrating the parallel execution of RNNs across multi-core CPU devices by scheduling computing pieces as soon as their data or control dependencies are fulfilled. The programmer does not need to define explicitly the parallel execution schedule at the source code level as it is managed on run-time by the system software, which makes it possible to apply W-Par on a wide range of parallel hardware with different core counts, cache hierarchies, or CPU clock frequencies. Programming environments like OmpSs [23] or OpenMP [20] support the definition of input and output dependencies and the dynamic management of them, which can be used to implement W-Par and to use it in a wide range of parallel computing scenarios.

This paper makes the following contributions over the state-of-the-art:

- We propose W-Par, a parallel execution model for RNNs and its variants LSTMs and GRUs. W-Par exploits model parallelism and it relies on source-code annotations of input and output dependencies across different RNNs compute kernels. W-Par can be applied to a wide range of parallel scenarios without requiring any change at the source code level.
- We carry out an extensive evaluation campaign considering data and model parallelism, a wide range of model parameters, and a state-of-the-art implementation of RNNs and its variants, LSTMs and GRUs. W-Par reaches performance speed-up up to 6.6× in comparison to the most recent implementation of the state-of-the-art Keras [18] deep learning framework, backed by Tensorflow 2.0 [10].

The rest of this paper is organized as follows: RNNs and its variants, LSTMs and GRUs, are described in Section 2. We describe the W-Par approach and how it exploits model parallelism on RNNs models in Section 3. Section 4 contains an exhaustive evaluation

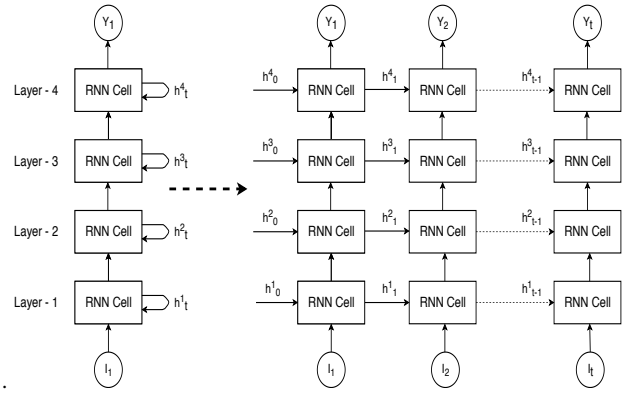


Figure 1: A 4-layer RNN represented in terms of directed cyclic graphs and corresponding unrolled RNN.

of W-Par and describes its specific experimental setup. Section 5 details the previous work done on parallelizing RNNs models on CPUs and GPUs. Finally, Section 6 provides a concluding remark with future directions for W-Par.

2 BACKGROUND ON RNN

RNNs adapt the standard feed-forward neural network model [14] to process sequential pieces of data like speech signals or text. RNNs have recurrent hidden states whose activation at a particular time is dependent on their values on previous times. More specifically, RNNs contain internal and time-dependent states that are updated following directed cycles of dependencies. These cycles influence the network activations by taking their values on previous time steps as inputs. At every time step, RNNs update their hidden state and predict by leveraging data corresponding to previous states, which improves the quality of its predictions. RNNs are widely used for sequence prediction [29] and classification tasks such as speech recognition or text analysis.

RNNs can be either expressed in terms of directed cyclic graphs, with their corresponding recurrent connections or via unfolded representations. The process of unfolding a recurrent network requires the removal of all cycles to form a directed acyclic graph [25]. Figure 1 represents a 4-layer RNN with recurrent connections. To unfold this model, we make a copy of each RNN cell for each time step with the corresponding input and output dependencies. Figure 1 shows the corresponding unfolded model, which does not have recurrent connections. The number of times we unroll RNN models is termed as *sequence length* or *unrolling length*. In Figure 1, t represents the sequence length.

Figure 2 shows a 4-layer deep RNN with a sequence length of 5. This RNN has 20 cells. Figure 2 also displays green arrows, which represent data dependencies involved in forward propagation, and red arrows, which represent dependencies for backward propagation. Data dependencies significantly constrain the order in which the different cells can be processed. For example, RNN cell 6 can only be processed once RNN cells 2 and 5 have finished updating their internal states. While Figure 2 displays a unidirectional model, it is also possible to train RNNs using a bidirectional paradigm [54], which has additional data dependencies. RNN cells are composed of either the Vanilla RNN, LSTM, or GRU structures, which are described in Sections 2.1, 2.2, and 2.3, respectively.

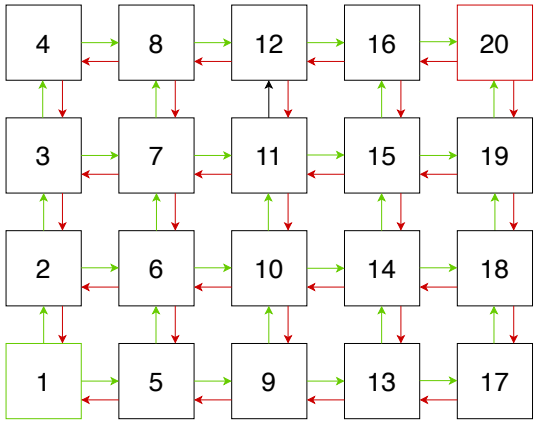


Figure 2: RNNs with 4 layers and 5 sequence length

2.1 Vanilla RNN

As the most basic RNN structure, the RNN or Vanilla RNN [24] adds a simple extension to feed-forward networks [14]. Its hidden states are updated from both the current input and the state of the previous time step.

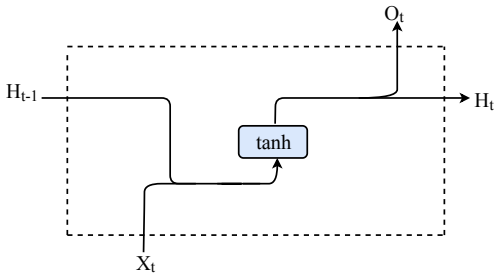


Figure 3: Vanilla RNN Cell Structure

Figure 3 shows the cell structure of Vanilla RNN. Equations 1 and 2 describe the update of Vanilla RNN cells and Table 1 defines the parameters of these equations.

$$H_t = tanh(W_h * [X_t, H_{t-1}] + B_h) \quad (1)$$

$$O_t = (W_o * H_t) + B_o \quad (2)$$

Vanilla RNNs are not able to capture long term dependencies on sequential data due to issues like the vanishing gradient problem [17]. Vanilla RNNs can make use of information in arbitrarily long sequences, but in practice, they are limited to looking back only a few steps. Two more advanced RNN architectures, the LSTM and the GRU networks aim to solve this problem. Both LSTMs and GRUs are an evolution of Vanilla RNN cells that maintain better long term dependencies and can be trained without suffering from gradient vanishing issues.

Table 1: Equation Parameters.

Parameter	Description
X_t:	Input vector
H_{t-1}:	Previous hidden state
H_t:	Current hidden state
W_h:	Weight Matrix for hidden state
W_i:	Weight Matrix for input gate
W_o:	Weight Matrix for output gate
W_f:	Weight Matrix for forget gate
W_u:	Weight Matrix for update gate
W_r:	Weight Matrix for reset gate
B_h:	Bias Matrix for hidden state
B_i:	Bias Matrix for input gate
B_o:	Bias Matrix for output gate
B_f:	Bias Matrix for forget gate
B_z:	Bias Matrix for update gate
B_r:	Bias Matrix for reset gate
C_{t-1}:	Previous cell state
C_t:	Current cell state
f_t:	Forget gate [4]
I_t:	Input gate [4]
O_t:	Output gate [4]
Z_t:	Update gate [4]
R_t:	Reset gate [4]
sigm:	Sigmoid Activation function [46]
tanh:	Tanh Activation function [46]

2.2 LSTM

The Long-Short Term Memory (LSTM) [27] is capable of modelling temporal sequences and their long-range dependencies accurately as compared to Vanilla RNN [24]. LSTMs are very widely used in the field of speech recognition [28], handwriting recognition [44], and speech synthesis [52]. Figure 4 presents the structure of the LSTM cell, which is the building block of any LSTM-based RNN model. It has special units called memory blocks, which contain memory cells with self-connections storing the temporal state of the network, along with special multiplicative units known as gates to control the information flow. Equations 3-8 define the update of LSTM cells [27], moreover, Table 1 defines the parameters of these equations. Operators $*$ represent matrix multiplications while operators $\odot$ and + represent element-wise multiplications and additions, respectively.

$$f_t = sigm(W_f * [X_t, H_{t-1}] + B_f) \tag{3}$$
$$I_t = sigm(W_i * [X_t, H_{t-1}] + B_i) \tag{4}$$
$$\bar{C}_t = tanh(W_c * [X_t, H_{t-1}] + B_c) \tag{5}$$
$$O_t = sigm(W_o * [X_t, H_{t-1}] + B_o) \tag{6}$$
$$C_t = f_t \odot C_{t-1} + I_t \odot \bar{C}_t \tag{7}$$
$$H_t = O_t \odot \tanh(C_t) \tag{8}$$

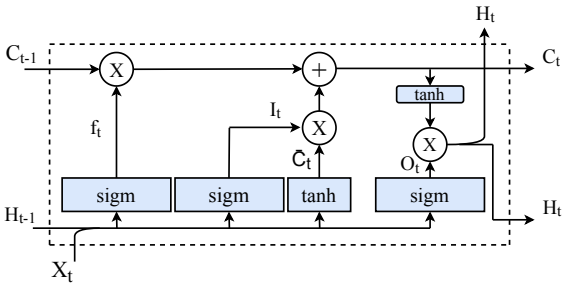


Figure 4: LSTM Cell Structure

2.3 GRU

The Gated Recurrent Unit (GRU) [19] is an evolution of the LSTM cell with a reduced number of parameters. They are faster than LSTMs and offer similar prediction accuracy. GRUs have gating units, which allow them to adaptively capture dependencies from extensive sequential data without discarding information from the earlier part of the sequence. Figure 5 shows a GRU cell structure, which is remarkably more straightforward than the LSTM cell that Figure 4 displays. Equations 9-12 define GRU cell computations and Table 1 defines their corresponding parameters. GRUs are an evolution of Vanilla RNNs widely used for automatic speech recognition [50]. Both GRUs and LSTMs achieve similar accuracy results.

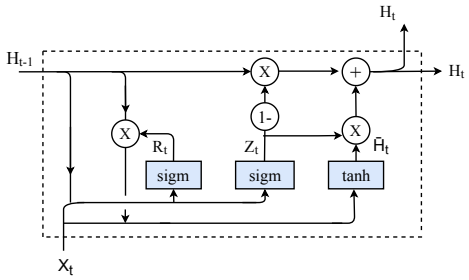


Figure 5: GRU Cell Structure

$$Z_t = sigm(W_z * [X_t, H_{t-1}] + B_z) \tag{9}$$
$$R_t = sigm(W_r * [X_t, H_{t-1}] + B_r) \tag{10}$$
$$\bar{H}_t = tanh(W_h * [X_t, R_t \odot H_{t-1}] + B_h) \tag{11}$$
$$H_t = Z_t \odot \bar{H}_t + (1 - Z_t) \odot H_{t-1} \tag{12}$$

2.4 Computational Graph

In parallel computing, the computational graph is a common way to represent computation tasks and their dependencies [37]. A computational graph is a directed acyclic graph (DAG) with each computation node representing an operation like matrix multiplication, a convolution, or an element-wise multiplication or addition. Computational graphs define the execution order of the different numerical kernels that compose a parallel workload. In Figure 6, a directed edge pointing from node X to node Y and from W to Y represents that node Y is dependent on nodes X and W, i.e. the

output of X and W constitute part of the node Y input. In addition, node Z depends on nodes Y and B.

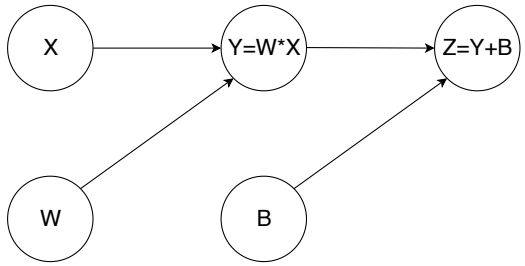


Figure 6: Computational graph with 5 computation nodes

Deep learning frameworks such as TensorFlow [10], MXNet [16] and Caffe [35] exploit computational graphs to represent deep learning workloads such as Vanilla RNN, LSTM and GRU [30]. Keras [18] is a high-level Application Programming Interface (API) for neural networks that run on top of TensorFlow or other packages. As such, it also exploits computational graphs. Keras [18] is widely used in software industries such as CERN [5], Uber [7], Google, Microsoft, Amazon Web Services [9].

The training and inference of a deep learning model is essentially the execution of its corresponding computational graph. Either training or inference of neural networks may rely on computational graphs to orchestrate their execution. For example, during training, the gradients of the loss function are computed by taking the weights and the biases as input. The weights are updated using gradient descent. A computational graph can efficiently express this workload. By default, the execution engines of the existing deep learning frameworks execute the computational graph in sequence according to its topological order. One training iteration of a batch consists of a single complete execution on the graph.

3 W-PAR APPROACH

We propose W-Par (Wavefront-Parallelization), an approach to parallelize multi-layer RNNs [57]. W-par conceives RNNs forward and backward propagation as a computational graph where nodes represent computation and edges identify data and control dependencies across them. W-Par exploits model parallelism on multi-layer RNNs network by creating multiple parallel tasks and specifying at the source code level their data or control dependencies. A runtime system software orchestrates the parallel execution by taking into account dependencies across different computing routines and scheduling them across multi-core CPU devices. W-Par relies on the basic structure of multi-layer RNNs, where a cell on a particular layer depends on the previous cell of the same layer and its counterpart cell of the previous layer.

The unrolled representations of RNNs, LSTMs and GRUs are composed of cells, which contain different state parameters like weights, biases, hidden states, etcetera. Table 2 represents the total number of states contained in single LSTM, GRU, and Vanilla RNN cells. To keep the information belonging to a single Vanilla RNN cell and its corresponding update operations, which are defined in equations 3-8, 13 different states are required. A 20-cell network composed of 4 layers of Vanilla RNN cells with a sequence length of 5 requires 199 states. Since cells belonging to the same layer share some of their states, we need 199 of them to represent a 20-cell network instead of 260. To represent a single LSTM cell and its update equations 3-8, we require 43 states. Similarly, to represent a single GRU cell, 34 states are required.

Table 2: Number of states required to represent Vanilla RNN, LSTM, and GRU cells.

Model	1 Cell	20 (4x5) Cells
Vanilla RNN	13	199
LSTM	43	639
GRU	34	523

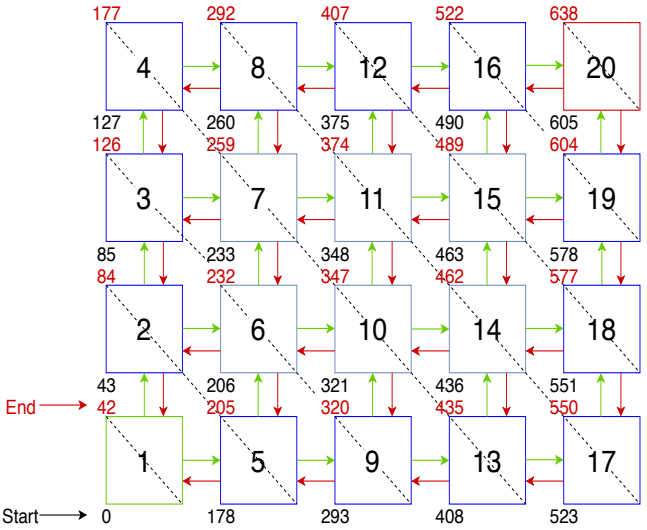


Figure 7: 4 layer with 5 sequence length unrolled deep LSTMs network with initial and final computational graph indexes per each cell

Figure 7 represents an unrolled LSTMs network with 4 layers and a sequence length of 5 with the corresponding initial and final state numbers per each cell. State-of-the-art RNN forward propagation compute cell outputs in the sequential order displayed in Figure 2, which implies processing first Cell 1, then Cell 2, until Cell 20. W-Par maps each cell computations in a single sequential task and orchestrates the parallel run taking into account dependencies across tasks, which defines a wavefront parallel scheme where Cell 1 is processed first, which produces the input dependencies consumed by Cells 2 and 5. For the case of LSTM cells, the 43 states of each cell and its corresponding updates are encapsulated within a single sequential task. Similarly, backward propagation can be parallelized by performing the update of Cell 20 as the starting sequential task.

Table 3 explicitly shows the input (In-dep) and output (Out-dep) dependencies for each cell in forward and backward propagation for the RNN example displayed in Figure 7. The input dependency (In-Dep) expresses the cells on which the execution of the current cell

Table 3: RNNs Cells forward and backward propagation dependencies

	Forward propagation		Backward Propagation	
Cell	In-Dep	Out-Dep	In-Dep	Out-Dep
1	-	2,5	2,5	-
2	1	3,6	3,6	1
3	2	4,7	4,7	2
4	3	8	8	3
5	1	6,9	6,9	1
6	2,5	7,10	7,10	2,5
7	3,6	8,11	8,11	3,6
8	4,7	12	12	4,7
9	5	10,13	10,13	5
10	6,9	11,14	11,14	6,9
11	7,10	12,15	12,15	7,10
12	8,11	16	16	8,11
13	9	14,17	14,17	9
14	10,13	15,18	15,18	10,13
15	11,14	16,19	16,19	11,14
16	12,15	20	20	12,15
17	13	18	18	13
18	14,17	19	19	14,17
19	15,18	20	20	15,18
20	16,19	-	-	16,19

relies and output dependency shows the cells which will rely on the current cell. Figure 8 shows a task dependency graph corresponding to the RNN shown in Figure 7. As we can see, cells belonging to the same depth level can be computed in parallel once their parent cell output is available. Similarly, for backward propagation, the dependency graph starts from the last cell processed in forward propagation. For a 4-layer RNN with a sequence length of 5, the maximum number of tasks running in parallel is 4. In general, for a N-layer RNN model with a sequence length of M, the maximum degree of parallelism is $minimum(N, M)$. Figure 8 displays data dependencies between the last tasks of the forward and backward propagation and their corresponding final task, which is focused on computing some average values. These dependencies enforce that there is no accuracy loss in W-Par as compared to the sequential version. Also, they enforce that no stale weights are used during the Training and Inference of the RNNs model.

3.1 Implementation of W-Par

The implementation of W-Par relies on the OmpSs [23] programming model. We introduce source code annotations to encapsulate sequential pieces of work that correspond to the update of a single RNN cell. We denote these sequential pieces of code as *tasks*. Our annotations specify all input and output dependencies per each task. Our W-Par code runs on the top of run-time system software that dynamically generates the task dependency graph by exploiting source code annotations. The run-time system maintains a list of ready to be executed tasks and assigns them to free CPU cores as soon as they become available. This execution model makes our W-Par implementation independent of hardware parameters like the number of cores, or the CPU clock frequency.

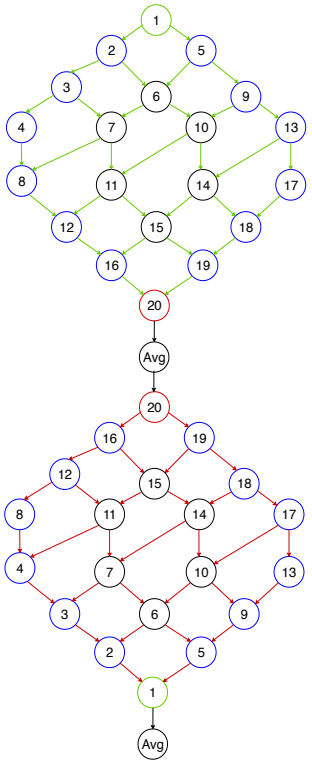


Figure 8: Forward and Backward propagation dependency graph

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
CG_Start	0	43	85	127	178	206	233	260	293	321	348	375	408	436	463	490	523	551	578	605
CG_End	42	84	126	177	205	232	259	292	320	347	374	407	435	462	489	522	550	575	604	638

Figure 9: Initial and final state indexes per cell for a 4-layer LSTM network with sequence length of 5

Algorithm 1 displays a pseudo-code representation of the W-Par Forward Propagation. The *Total_Cells* parameter represents the total number of RNN cells. Figure 7 shows one example with a *Total_Cells* parameter equal to 20. The *CG_Start* and *CG_end* arrays contain the initial and the final indexes of all states mapped at each cell, respectively. Figure 9 shows an example of these arrays corresponding to the network displayed in Figure 7. Depending on the number of input and output dependencies of each cell, the corresponding *forward_propagation* task is instantiated by a different pragma annotation. W-Par applies the same approach to Backward Propagation by launching parallel tasks based on the input and output dependencies of the corresponding cell. The last cell of the forward pass is the first one of the backward propagation, as Figure 7 shows. During backward propagation, the states marked in red and black in Figure 7 represent the initial and the final states, respectively, assigned to each sequential task.

Algorithm 1 Forward Propagation

```
1: Total_Cells ▷ For a 4-layer RNN with a sequence length of 5, the total number of cells is 20
2: a ▷ Computation graph
3: CG_Start ▷ Array containing initial state indexes
4: CG_End ▷ Array containing final state indexes
5: cell = -1 ▷ current cell being processed
6: for U ← 0 to seq.length do ▷ traverse through sequence length
7:     for L ← 0 to layers do ▷ traverse through layers
8:         s =CG_Start[U∗layers + l] ▷ Starting index of a cell
9:         e =CG_End[U∗layers + l] ▷ End index of a cell
10:        + + cell
11:        if u > 0 and l> 0 then
12:            #pragma omp task in(a[s − 1], a[CG_End[cell − layers]]) out(a[e]) ▷ Task with 2 In-Dep and 2 Out-Dep
13:            forward_propagation(s, e, a)
14:        end if
15:        if u = 0 and l> 0 then
16:            #pragma omp task in(a[s−1]) out(a[e]) ▷ Task with 1 In-Dep and 2 Out-Dep
17:            forward_propagation(s, e, a)
18:        end if
19:        if u > 0 and l= 0 then
20:            #pragma omp task in(a[CG_End[cell−layers]]) out(a[e]) ▷ Task with 1 In-Dep and 1 Out-Dep
21:            forward_propagation(s, e, a)
22:        end if
23:        if u = 0 and l= 0 then ▷ Starting node in computation graph
24:            #pragma omp task in(a[s]) out(a[e]) ▷ Task with 1 In-Dep and 1 Out-Dep
25:            forward_propagation(s, e, a)
26:        end if
27:    end for
28: end for
29: Task wait ▷ Wait for all Forward propagation tasks to finish
```

4 EVALUATION

4.1 Experimental setup

We conduct our experiments on a 48-core system composed of two 24-core Intel Xeon Platinum 8160 processors at 2.1 GHz with SuSe Linux OS with the following cache storage: L1D 32K; L1i 32K; L2 cache 1024K; L3 cache 33792K. We use the C programming language-based KANN framework [3] to build the W-Par approach. We extend the KANN framework with the OmpSs [23] programming model. The OmpSs runtime system is in charge of dynamically scheduling task instances to the compute units. We compile all RNNs models with GCC 8.1.0 with -O3 optimization flag and use the sequential Intel MKL library 2019.04. Each experiment consists of 200 repetitions. We do not consider the first set of 100 repetitions when we measure performance, to avoid measurement noise. Our performance metric is the mean execution time in milliseconds [ms].

We compare W-Par against the Keras 2.3.1 [18] framework backed by Tensorflow 2.0.0 [10]. Keras experiments run using python 3.6.4 with the Intel MKL library 2019.04, which can run on multiple threads. We use the Intel optimized TensorFlow [8] installation for our experiments. Tensorflow relies on Intel MKL-DNN primitives. Keras performance is optimized using the widely supported Intel suggestions for thread parallelism [6]. For both W-Par and Keras experiments, we map one thread per core. We use for both Keras and W-Par experiments the same RNN initial configuration. We extend the Keras implementation of the LSTM benchmark [15] for our work. We initialize all the biases to one, and we use a random uniform distribution for the weights, no layer normalization, stateful implementation, and a dropout layer for input. We use the RmsProp [33] optimizer. In all our experiments, W-Par and Keras implementations display very similar accuracy.

Our evaluation also considers an approach implemented with KANN that entirely relies on data parallelism. We call this approach *Seq*. Seq splits batches of samples into mini-batches that are processed in parallel. Seq only relies on data-parallelism and processes each mini-batch sequentially. W-Par relies on both data- and model-parallelism, which means that it can split a batch into several mini-batches and process each mini-batch in parallel.

Data-set: We consider three different data-sets in our evaluation: The TIDIGITS speech corpus data-set [41], the real-world text-corpus Wikipedia data-set [31, 55] and synthetic data. TIDIGITS contains speech which was originally designed and collected to evaluate algorithms for speaker-independent recognition of connected digit sequences. Section 4.2 contains results considering the TIDIGITS data-set. Wikipedia is a data-set of 1.4 billion words. We consider the next character prediction problem when employing the Wikipedia data-set. Synthetic data is composed of 4-digit numbers generated from a randomly sampled uniform distribution $[1, 10^4]$. The long binary addition and multiplication problems [34, 36, 45] are considered when using synthetic data. These two problems are commonly used for evaluating the viability of new RNN designs, and usually, the numbers to be added or multiplied are explicitly encoded by a single input vector at each time step. Section 4.3 contains results considering the Wikipedia and the Synthetic data-sets.

4.2 Performance on Speech Recognition Tasks

In this section we provide an evaluation of W-Par against both Keras and Seq considering a speech recognition task on the TIDIGITS [41] data-set. We present general results considering different model parameter settings. We also evaluate in detail the impact on performance of key parameters.

Speed-up on CPUs: W-Par is several times faster than Keras for a wide range of RNN models and configurations on CPUs. Table 4 displays single batch training time for 6-layer RNN models composed of LSTMs, GRUs, and Vanilla RNNs using the Seq and W-Par approaches. Training time includes the forward and backward propagation plus the gradient update time per batch. The first columns present the configuration values of the considered models in terms of input dimension, hidden dimension (number of neurons), batch size, and sequence length (unrolling length). These configurations represent similar workloads belonging to the LSTM benchmark [15] and DeepCPU benchmark [58]. Table 4 reports

Keras, Seq and W-Par training times. The speed-up is reported for W-Par in comparison to Keras RNNs models Training time. We let Keras select appropriate degrees for inter-op and intra-op [10] parallelism. Our results show that W-Par significantly outperforms Keras and Seq, with speed-up in range of 1.5x to 4.7x.

Speed-up on GPUs: We consider the same configurations of 6-layer LSTMs and GRUs models as Table 4 on a Tesla V100 SXM2 16 GB GPU using Tensorflow version 2.1.0. Table 5 shows the training time speed-up for W-Par running on the multi-core CPU described in Section 4.1 against Keras on the V100 GPU. For small batch-sizes W-Par improves Keras execution time but, as model parameters increase their values, Keras executions outperform W-Par since GPU many-cores efficiently compute large matrix-matrix multiplications. Similar results are obtained for inference.

Next, we look into the 5 relevant parameters that impact on the performance of RNNs models training and inference time: the size of the hidden state, batch size, number of cores, mini-batch size and number of layers. To demonstrate the significance of W-Par, we conduct the next experiments considering both 8-layer and 12-layer LSTM models keeping the sequence length 100 and input size as 256, unless explicitly stated otherwise. For all experiments, the best execution times are reported when we execute Keras, W-Par, and Seq on core counts 1, 2, 4, 8, 16, 24, 32, 48.

Varying number of cores and mini-batch size training: We evaluate W-Par and Seq considering mini-batch sizes [43] of 1, 2, 4, and 6 on different core counts. Gradients are computed considering contributions from all mini-batches. Figure 10 and Figure 11 show the speed-up of W-Par and Seq, respectively, while using different mini-batch sizes. Categories mbs:1, mbs:2, mbs:4, and mbs:6, represent the speed-up of W-Par when using mini-batch sizes of 1, 2, 4, and 6 respectively against Seq using a mini-batch size of 1.

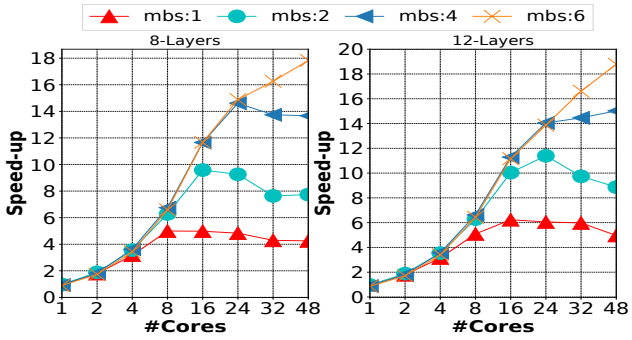


Figure 10: W-Par speed-up against Seq-mbs:1. Batch size 128.

For both W-Par and Seq, we achieve the best performance when training both 8- and 12-layers networks with mbs:6. It is the configuration that exposes more parallelism to the underlying parallel hardware since it splits the 128 samples mini-batch into 6 different subsets that can be processed in parallel. The performance benefits of this data-parallel approach correspond to the one's displayed by Seq in Figure 11. Additionally, W-Par parallelizes backward and forward propagation's performed over the samples of each subset, which is the reason why W-Par runs much faster than Seq in general, as Figures 10 and 11 show.

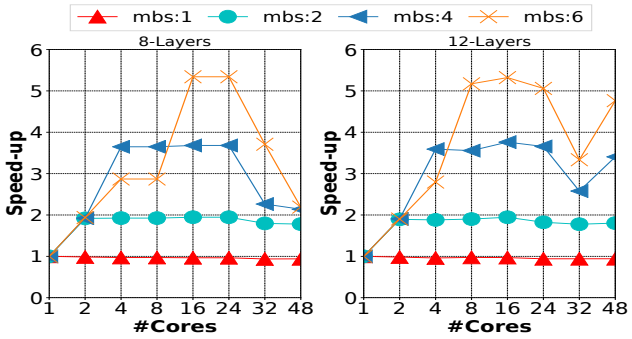
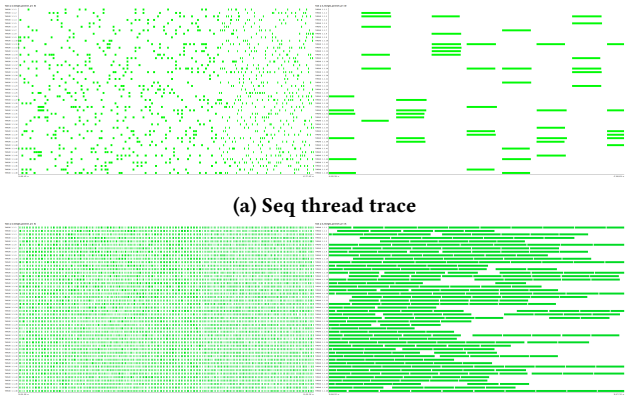


Figure 11: Seq speed-up against Seq-mbs:1. Batch size 128.

For both Seq and W-Par, 8- and 12-layer scalability decreases when going from 24 to 32 cores. This effect is due to Non-Uniform Memory Access (NUMA) effects, which appear when we need to use the two sockets of our experimental platform. For executions on less than 24 cores, a single socket can be used and thus there are no NUMA performance issues. The only approach that increases its performance when going from 24 to 32 cores is W-Par mbs:6. The substantial amount of concurrency exposed by this configuration takes advantage of the additional cores and provides additional performance despite NUMA effects.



(a) Seq thread trace

(b) W-Par thread trace

Figure 12: Parallel training of an 8-layer LSTM on 48 cores.

Figure 12 displays parallel executions corresponding to both W-Par and Seq when training an LSTM 8-layer model using a mini-batch size of 6. The x-axis displays time and the y-axis represents the different cores involved in the parallel execution. For specific time and core, Figure 12 displays either green or white color. Green means that the core is running useful work while the white color encodes idle time. For both Seq and W-Par, we show a general view of the execution on the left-hand side plot and a more detailed representation of a specific time interval on the right-hand side plot. For the case of Seq, Figure 12a displays large white areas, which indicates idle CPU time. In the fine-grain representation displayed in the right-hand side, we see how Seq never uses more than 6 cores, which corresponds to the maximum parallelism allowed by

Table 4: Training times and speed-up of LSTMs, GRUs and RNN, comparing W-Par, Seq and Keras on CPU

Model Parameters				LSTM exec. time (ms)			GRU exec. time (ms)			RNN exec. time (ms)			Speed-up		
Input	Hidden	Batch size	Seq. Len.	KERAS	Seq	W-Par	KERAS	Seq	W-Par	KERAS	Seq	W-Par	**LSTM**	**GRU**	**RNN**
64	256	128	100	1,983.8	1,601.2	436.1	1,343,4	975.4	349.1	341.1	365.5	118.6	4.5	3.9	2.9
256	256	128	100	1,966.3	1,669.0	461.9	1,340.5	962.2	323.3	345.1	381.6	110.0	4.3	4.2	3.5
1024	256	128	100	2,053.7	1,846.3	624.5	1,376.1	1,089.1	444.6	377.2	406.7	131.1	3.3	3.1	2.9
256	256	1	2	9.43	7.7	6.4	8.4	5.3	5.2	3.6	1.8	2.0	1.5	1.7	1.9
256	256	1	10	31.3	20.2	10.0	35.8	7.0	7.6	8.9	6.5	4.0	3.1	4.7	2.2
256	256	1	100	312.5	237.9	66.8	362.8	160.3	47.7	76.2	59.5	28.8	4.7	7.6	2.6
64	256	256	100	2,972.1	2,788.5	821.4	2,129.9	1,687.2	598.1	620.1	683.0	201.1	2.4	3.6	3.1
64	1024	256	100	28,108.6	31,539.9	8,852.1	22,014.2	19,490.6	6,534.5	6,476.4	6,889.6	2,247.4	3.2	3.4	2.9
256	256	256	100	2,986.8	2,796.5	694.0	2,140.1	1,623.2	527.9	623.7	663.8	184.7	4.3	4.1	3.4
256	1024	256	100	28,157.6	42,245.5	9,051.1	21,963.3	19,734.9	6,504.2	6,483.7	7,617.1	2,081.4	3.1	3.4	3.1
1024	256	256	100	3,077.6	3,021.4	903.3	2,193.1	1,793.5	650.8	684.5	730.6	226.4	3.4	3.4	3.1
1024	1024	256	100	28,220.4	31,444.4	8,992.8	22,039.2	23,429.2	6,619.3	6,558.4	7,078.6	2,078.7	3.1	3.3	3.2

Table 5: Training times and speed-up of LSTMs and GRUs, comparing W-Par on CPU and Keras on GPU

Model Parameters				LSTM exec. time (ms)		GRU exec. time (ms)		Speed-up	
Input	HiddenSize	Batch Size	Seq Len	Keras-GPU	W-Par-CPU	Keras-GPU	W-Par-CPU	**LSTM**	**GRU**
64	256	128	100	641.27	436.10	556.53	349.15	1.47	1.59
256	256	128	100	660.32	461.95	531.04	323.30	1.43	1.64
1024	256	128	100	622.33	624.57	572.70	446.65	1.00	0.37
256	256	1	2	22.14	6.40	18.15	5.23	3.46	3.47
256	256	1	10	67.78	10.09	66.46	7.63	6.72	8.72
256	256	1	100	607.89	66.86	571.09	47.70	9.09	11.97
64	256	256	100	651.71	821.44	589.70	598.13	0.79	0.99
64	1024	256	100	870.78	8,852.11	748.52	6,534.56	0.10	0.11
256	256	256	100	640.98	694.08	566.50	527.91	0.92	1.07
256	1024	256	100	887.27	9,051.15	759.18	6,504.25	0.10	0.12
1024	256	256	100	646.65	903.34	600.79	650.86	0.72	0.92
1024	1024	256	100	928.20	8,992.87	776.96	6,619.37	0.10	0.12

a mini-batch size of 6. For the case of W-Par, the fine-grain plot of Figure 12b displays a much more intense use of the CPUs, which corresponds to the large degree of concurrency exposed to the hardware by W-Par. The maximum degree of parallelism that W-Par can expose for this 8-layer LSTM model with a sequence length of 100 is 8, as Section 3 indicates. Since each batch is split into 6 mini-batches that can be processed independently, the maximum degree of parallelism, in this case, is 48.

When the number of available cores is smaller than the maximum possible parallelism, W-Par achieves good performance because of it exposes more parallelism than the one the hardware can consume. In this case, parallelism is handled in the same way as having more cores than parallel tasks, that is, W-Par lets the runtime system assign ready tasks to cores as they become available. For example, Figure 10 shows an experiment where the number of available cores is 48, and for the mbs:6 configuration, the maximum available parallelism is 72 (12x6). The mbs:6 configurations outperform all the other scenarios.

Varying hidden/batch size: In Figure 13 we display Seq and W-Par training time speed-up for the 8-layer and 12-layer RNN LSTM models with a batch size varying from 128 to 1024 in a combination of 128 and 256 long-hidden states comparing to Keras. W-Par achieves very significant speed-up within the $4.0-6.5\times$ range for all configurations. The difference between W-Par and Keras is smaller for configurations using the largest hidden and batch sizes since the parallel MKL implementation that Keras uses provides the best performance in these scenarios. Importantly, W-Par beats Keras even in these scenarios. Seq can significantly beat Keras just for configurations using 128 and 256 batch sizes and a hidden size of 128. When using 512 and 1024 batch sizes, Keras performance improves and almost matches Seq. This effect is explained by the good performance in these scenarios of the parallel MKL implementation that Keras uses.

Varying number of layers: Figure 14 displays the speed-up achieved for training and inference by W-Par and Seq in comparison to Keras while varying the number of layers. Both the batch size and the hidden layer size parameters are set to 128. W-Par training time

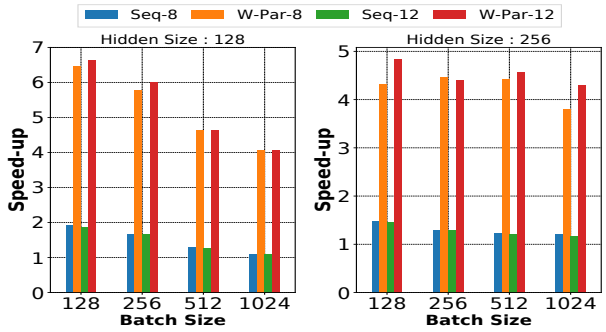


Figure 13: W-Par and Seq speed-up against Keras

speed-up grows as the number of layer increases for both training and inference time since the more layers the model have, the more significant is the parallelism exposed by W-Par. For all layer counts, W-Par exposes more parallelism than Keras and outperforms it. For a 12-layer LSTM model, the speed-up achieved by W-Par is 6.6× and 5.4× for inference and training, respectively. The parallelism that Seq exposes is independent of the number of layers of the model. The speed-up of Seq against Keras is the same for all the considered layer counts, therefore, Keras does not expose more parallelism with larger layer counts.

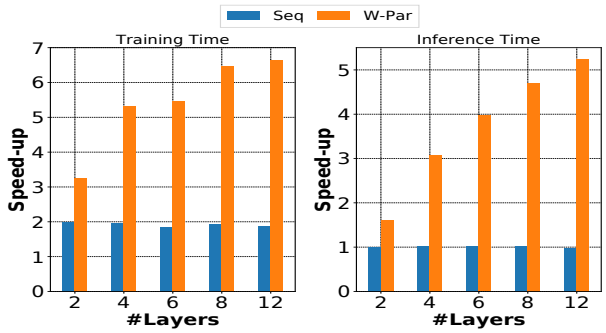


Figure 14: Multi-layer W-Par and Seq speed-up against Keras

Task and Core Complexity: For an LSTM model with hyperparameters Seq Length=100, Batch Size=128, Input Size=256, Hidden size=512, and mbs=6, the working set size of each LSTM cell, which is processed by one task, is 4.71 MB. The LSTM cell working set size does not fit in the cache hierarchy of our experimental platform, which is described in Section 4.1. Table 6 shows the average work done by each core and task on 48-cores executions. As we increase the number of layers, data per core increases since the model becomes larger, and more tasks run on each core. Task time decreases as we increase the number of layers because the increasing amount of concurrency exposed to the multi-core architecture reduces the idle time of the parallel run. For 12 layers, there is an increase in task time due to a small increase in the overhead to manage the parallel workload.

Table 6: LSTM model task and core complexities

Layers	Time/Task (ms)	Time/Core (ms)	Data/Core (MB)
2	0.374	18.817	19.660
4	0.267	26.809	39.321
6	0.239	35.959	58.982
8	0.237	47.642	78.643
12	0.250	74.948	117.964

4.3 Performance on Next Character Prediction, Addition and Multiplication Tasks

In this section, we compare Keras and W-Par for the next character prediction, addition and multiplication tasks.

Next Character prediction task: We consider next character prediction on a real-world Wikipedia text-corpus [31, 55] where RNN models are commonly used. The complete set is about 1.4 billion characters long. Each character is represented by one-out-of-N coding. We used 95 of the most common characters (including small letters, capitals, numbers and punctuation), and one 'unknown' character used to map any character not part of the 95 common ones. We run experiments considering the Keras and W-Par approaches. RNN models are used here to predict the probability distribution of the next character given a sequence of text. The models use unrolling length of 100. Figure 15 reports the speed-up of W-Par for LSTMs, GRUs and Vanilla RNNs against Keras. We consider batch sizes of 64, 128, 256, 512, 1024, and hidden state sizes of 128 and 256. W-Par achieves a maximum speed-up of 3.69×, 5.49×, 6.10×, and 5.88× for 2, 4, 8, and 12 layers, respectively, with respect to the Keras across all configurations in Figure 15.

Addition Task: We also evaluate our models considering the addition task [34, 45]. For this task, the length of the input depends on the numerical range of the input set and, therefore, determines the difficulty of the addition prediction since larger inputs imply larger numbers. We consider the addition of two 4-digit numbers of chosen uniformly from $[1, 10^4]$. Figure 16 displays the results of our evaluation. W-Par achieves a maximum speed-up of 3.51×, 4.26×, 4.27×, and 4.38× for 2, 4, 8, and 12 layers, respectively, with respect to Keras across all configurations.

Multiplication Task: The multiplication task [36] is similar to the addition task. We compare the output against the multiplication of two 4-digit numbers. Figure 17 shows how W-Par achieves a maximum speed-up of 3.64×, 4.86×, 5.09×, and 6.21× for 2, 4, 8, and 12 layers, respectively, with respect to Keras across all configurations.

5 RELATED WORK

A large amount of work has been devoted in the previous years to accelerate RNNs [53]. One of the most commonly applied approaches is to accelerate RNNs by using GPU accelerators [40]. These approaches are useful although they typically rely on data parallelism instead of model parallelism, as we do in this paper.

Expressing parallel workloads as directed acyclic graphs where edges represent control or data dependencies and nodes represent computation pieces dates back to the data flow computation concept [21] developed in the 1960s. This approach mostly targets

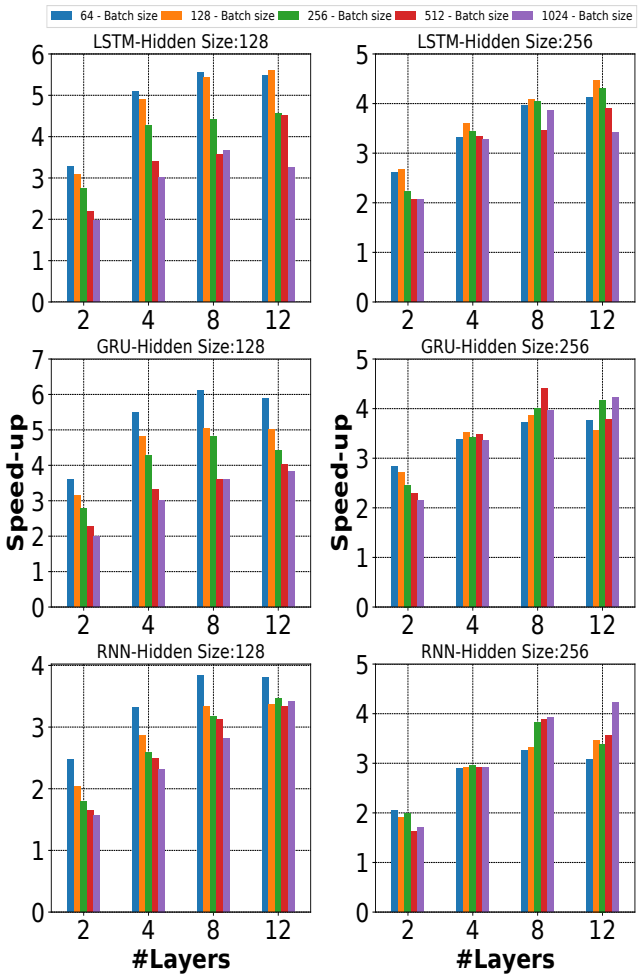


Figure 15: Next Character task

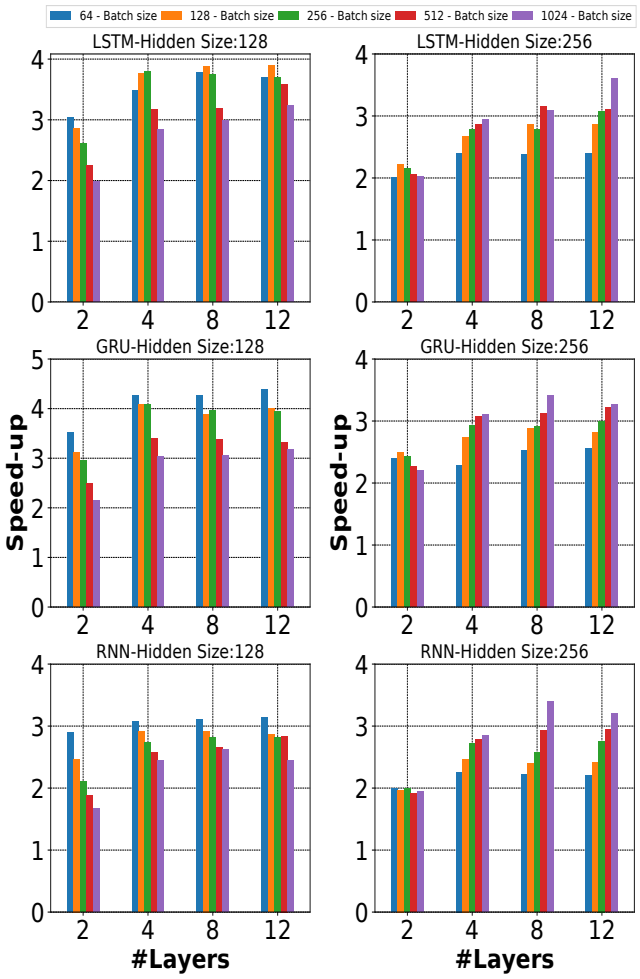


Figure 16: Addition task

performance improvements via compiler or run-time system optimizations [1, 47, 48]. W-Par leverages this previous idea and applies it to Multi-layer RNNs on many-core CPUs architecture.

W-Par relies on wavefront parallel patterns, which also appear in critical scientific applications such as sequence alignment [11] or dynamic programming [56]. This concept was proposed several decades ago [42]. Previous research work [12, 22] has explored the applicability of the task programming model in the parallelization of wavefront patterns.

One key factor affecting the performance of conventional multi-layer RNNs models are the dependency among the cells in both the same layer as well as the next layer. Dependency among cells is the fundamental aspect that W-Par exploits. Previous work targeting RNNs acceleration on GPUs very briefly describe this idea but do not provide specific details on how to use it [13]. It is unclear how to exploit the irregular parallelism of W-Par in the context of GPUs, where parallel executions are composed of a large number of warps, which are groups of threads executing the same instruction on different pieces of data.

Some previous approaches [49] aim at increasing the performance of LSTM models by leveraging three techniques: multiprogramming, model parallelism, and a helper core to accelerate matrix-vector products. These approaches do not fully exploit model parallelism since they apply a static scheduling approach in a 2-core design implemented in an FPGA. They achieve a 1.75× speed-up with respect to a single core execution by exploiting model parallelism, while W-Par achieves linear speed-up when running on two and four cores. Overall, these approaches focus on FPGA acceleration, while W-Par squeezes all the available performance that model parallelism can deliver.

Deep learning frameworks such as TensorFlow [10], MXNet [16] or Caffe [35] mostly express the computation of deep learning models in terms of computational graphs [37]. TensorFlow uses both the Eigen Library [2] and OpenMP [20], each with their thread pool, which results in more software threads than available physical cores. This is a conceptually simple and feasible approach for neural networks since they are in general able to expose a large amount of parallelism within each thread pool. However, performance issues

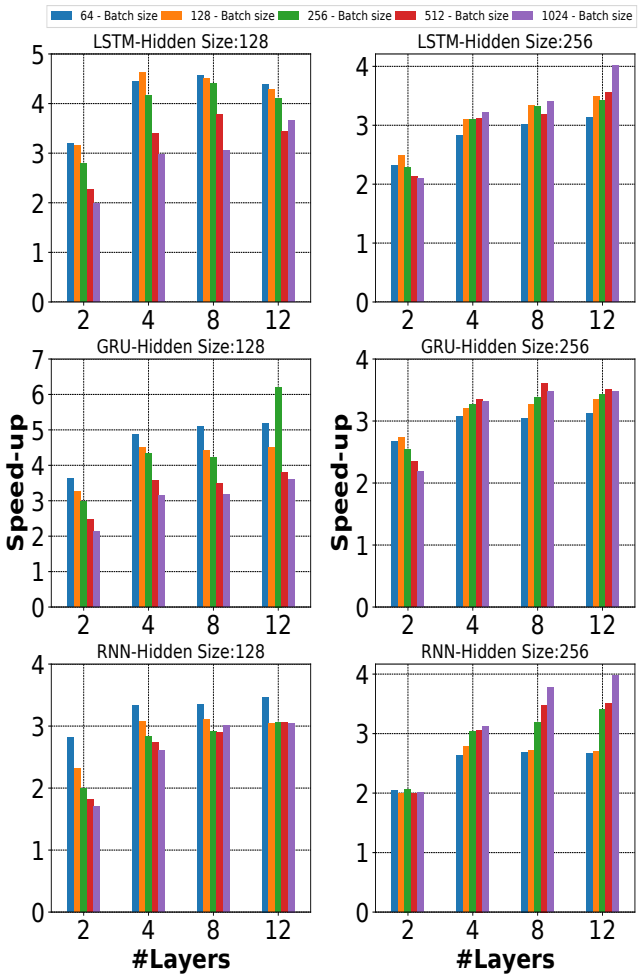


Figure 17: Multiplication task

due to thread migration and interference often result in sub-optimal performance, especially when large core counts are involved in the parallel run.

Previous work reports results on LSTMs parallel execution time using well-known software tools like Tensorflow, Keras, or PyTorch [15]. This previous work relies entirely on data parallelism to accelerate RNNs workloads. Model parallelism is not used in these research activities. Our paper compares W-Par with some of these software tools and demonstrates its superior performance.

Some previous approaches characterize RNNs workloads and identify low data reuse as one of the main factors causing low performance [58]. Techniques like cache partitioning are used to improve data reuse at the cache hierarchy level. The authors of [58] achieve very significant performance improvements by improving data reuse. These approaches based on improving data reuse of RNNs in the context of multi-core CPU devices are orthogonal to our approach and can be combined with it.

Previous work-study the effect of a hierarchy of recurrent neural networks on processing time series [32]. Each layer is a recurrent network that uses the hidden state of the previous layers as input. This architecture enables hierarchical processing for challenging tasks and captures the structure of the time series. This previous work achieves remarkable performance using simple training approaches and shows the importance of having multi-layer schemes [26], which can be easily parallelized by W-Par.

6 CONCLUSION

This paper shows that W-Par[1] is a convenient approach for accelerating training and inference of multi-layer RNNs models on multi-core CPU devices. It outperforms popular software frameworks like Keras or Tensorflow. Indeed, W-Par achieves up to 6.6× speedup with respect to the state-of-the-art on massive data-sets composed of the real text. Additionally, we present experimental evidence on the benefits of running RNNs on relatively large core counts, although NUMA effects may undermine performance on specific scenarios. This paper shows that RNN models training and inference can run on multi-core CPUs and achieve excellent performance, which complements the state-of-the-art approaches based on executing RNNs models on GPUs. Applying cache and memory reuse optimizations [58] on W-Par can potentially increase its benefits even more.

ACKNOWLEDGMENTS

This work has been supported by the European Union's Horizon 2020 research and innovation program (MB2020 project, grant agreement 779877), by the European HiPEAC Network of Excellence, by the Spanish Ministry of Economy and Competitiveness (contract TIN2015-65316-P), and by Generalitat de Catalunya (contracts 2017-SGR-1414 and 2017-SGR-1328). M. Casas has been partially supported by the Spanish Ministry of Economy, Industry, and Competitiveness under Ramon y Cajal fellowship number RYC-2017-23269.

REFERENCES

[1] 2016. Tiramisu Compiler. https://github.com/Tiramisu-Compiler/tiramisu. (2016).
[2] 2017. Eigen: a C++ Linear Algebra Library. http://eigen.tuxfamily.org/. (2017).
[3] 2017. KANN framework. https://github.com/attractivechaos/kann. (2017).
[4] 2017. Understanding LSTM Networks. https://colah.github.io/posts/2015-08-Understanding-LSTMs/. (2017).
[5] 2018. CERN-Districuted Keras. https://github.com/cerndb/dist-keras. (2018).
[6] 2018. Intel Keras performance improvement. https://intel.ly/2N0xZrE. (2018).
[7] 2018. Uber-horovod. https://github.com/horovod/horovod. (2018).
[8] 2018. Using the Intel optimized tensroflow. https://intel.ly/2RPWklu. (2018).
[9] 2018. Why use Keras? https://keras.io/why-use-keras/. (2018).
[10] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, and Ian Goodfellow et. al. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). http://tensorflow.org/ Software available from tensorflow.org.
[11] John Anvik, Steve MacDonald, Duane Szafron, Jonathan Schaeffer, Steven Bromling, and Kai Tan. 2002. Generating Parallel Programs from the Wavefront Design Pattern. DOI:https://doi.org/10.1109/IPDPS.2002.1016487
[12] John Anvik, Steve MacDonald, Duane Szafron, Jonathan Schaeffer, Steven Bromling, and Kai Tan. 2002. Generating Parallel Programs from the Wavefront Design Pattern. DOI:https://doi.org/10.1109/IPDPS.2002.1016487

[1] https://gitlab.com/robinkumarsharma/w-par

[13] Jeremy Appleyard, Tomás Kociský, and Phil Blunsom. 2016. Optimizing Performance of Recurrent Neural Networks on GPUs. *CoRR* abs/1604.01946 (2016). arXiv:1604.01946 http://arxiv.org/abs/1604.01946

[14] George Bebis and Michael Georgiopoulos. 1994. Feed-forward neural networks. *Potentials, IEEE* 13 (11 1994), 27 – 31. DOI: https://doi.org/10.1109/45.329294

[15] Stefan Braun. 2018. LSTM Benchmarks for Deep Learning Frameworks. *CoRR* abs/1806.01818 (2018). arXiv:1806.01818 http://arxiv.org/abs/1806.01818

[16] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. (12 2015).

[17] Kyunghyun Cho, Bart van Merrienboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. (2014). arXiv:cs.CL/1409.1259

[18] FranÃğois Chollet. 2015. keras. https://github.com/keras-team/keras. (2015).

[19] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Y. Bengio. 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. (12 2014).

[20] L. Dagum and R. Menon. 1998. OpenMP: An Industry-Standard API for Shared-Memory Programming. *Computational Science & Engineering, IEEE* 5 (02 1998), 46 – 55. DOI: https://doi.org/10.1109/99.660313

[21] Jack Dennis. 1980. Data Flow Supercomputers. *Computer* 13 (12 1980), 48– 56. DOI: https://doi.org/10.1109/MC.1980.1653418

[22] Antonio Dios, Angeles Navarro, Rafael Asenjo, Francisco Corbera, and Emilio Zapata. 2012. A case study of the task-based parallel wavefront pattern. 22 (01 2012). DOI: https://doi.org/10.3233/978-1-61499-041-3-65

[23] Alejandro Duran, Eduard AyguadÃĹ, Rosa M. Badia, JesÃžs Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. Ompss: a Proposal for Programming Heterogeneous Multi-Core Architectures. *Parallel Processing Letters* 21 (06 2011), 173–193. DOI: https://doi.org/10.1142/S0129626411000151

[24] Felix Gers and E. Schmidhuber. 2001. LSTM recurrent networks learn simple context-free and context-sensitive languages. *Neural Networks, IEEE Transactions on* 12 (12 2001), 1333 – 1340. DOI: https://doi.org/10.1109/72.963769

[25] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

[26] Ian Goodfellow, Yaroslav Bulatov, Julian Ibarz, Sacha Arnoud, and Vinay Shet. 2013. Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks. (12 2013).

[27] Alex Graves. 2013. Generating Sequences With Recurrent Neural Networks. (08 2013).

[28] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. 2013. Speech Recognition with Deep Recurrent Neural Networks. *ICASSP, IEEE International Conference on Acoustics, Speech and Signal Processing - Proceedings* 38 (03 2013). DOI: https://doi.org/10.1109/ICASSP.2013.6638947

[29] Trevor Hastie, Rob Tibshirani, and Jerome Friedman. 2009. The Elements of Statistical Learning: Springer. *Elements* 1 (01 2009).

[30] Ryan Hefron, Brett Borghetti, James Christensen, and Christine Kabban. 2017. Deep long short-term memory structures model temporal dependencies improving cognitive workload estimation. *Pattern Recognition Letters* 94 (05 2017). DOI: https://doi.org/10.1016/j.patrec.2017.05.020

[31] M. Hermans and B. Schrauwen. 2013. Training and analyzing deep recurrent neural networks. *Advances in Neural Information Processing Systems* (01 2013).

[32] M. Hermans and B. Schrauwen. 2013. Training and analyzing deep recurrent neural networks. *Advances in Neural Information Processing Systems* (01 2013).

[33] Geoffrey Hinton. 2017. RmsProp optimizer. https://bit.ly/36udFGJ. (2017).

[34] Sepp Hochreiter and JÃijrgen Schmidhuber. 1997. Long Short-term Memory. *Neural computation* 9 (12 1997), 1735–80. DOI: https://doi.org/10.1162/neco.1997.9.8.1735

[35] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *MM 2014 - Proceedings of the 2014 ACM Conference on Multimedia* (06 2014). DOI: https://doi.org/10.1145/2647868.2654889

[36] Lukasz Kaiser and Ilya Sutskever. 2016. Neural GPUs Learn Algorithms.

[37] Richard M Karp and Rayamond E Miller. 1966. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM J. Appl. Math.* 14, 6 (1966), 1390–1411.

[38] Hugo Larochelle, Yoshua Bengio, Jérôme Louradour, and Pascal Lamblin. 2009. Exploring strategies for training deep neural networks. *Journal of machine learning research* 10, Jan (2009), 1–40.

[39] Honglak Lee, Roger Grosse, Rajesh Ranganath, and Andrew Y Ng. 2009. Convolutional deep belief networks for scalable unsupervised learning of hierarchical representations. In *Proceedings of the 26th annual international conference on machine learning*. ACM, 609–616.

[40] Tao Lei, Yu Zhang, Sida Wang, Hui Dai, and Yoav Artzi. 2018. Simple Recurrent Units for Highly Parallelizable Recurrence. 4470–4481. DOI: https://doi.org/10.18653/v1/D18-1477

[41] R. G. Leonard and G. Doddington. 1993. Tidigits speech corpus. *Texas Instruments, Inc* (1993).

[42] E Lewis and Lawrence Snyder. 1970. Pipelining Wavefront Computations: Experiences and Performance. *Lecture Notes in Computer Science* 1800 (02 1970). DOI: https://doi.org/10.1007/3-540-45591-4_35

[43] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander Smola. 2014. Efficient mini-batch training for stochastic optimization. *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (08 2014). DOI: https://doi.org/10.1145/2623330.2623612

[44] Marcus Liwicki, Alex Graves, and Horst Bunke. 2019. A Novel Approach to On-Line Handwriting Recognition Based on Bidirectional Long Short-Term Memory Networks. (10 2019).

[45] Dan Neil, Michael Pfeiffer, and S-C Liu. 2016. Phased LSTM: Accelerating Recurrent Network Training for Long or Event-based Sequences.

[46] Chigozie Nwankpa, Winifred Ijomah, Anthony Gachagan, and Stephen Marshall. 2018. Activation Functions: Comparison of trends in Practice and Research for Deep Learning. (2018). arXiv:cs.LG/1811.03378

[47] K.J. Ottenstein and L.M. Ottenstein. 1984. The program dependence graph in a software development environment. *SIGSOFT Softw. Eng. Notes* 19 (01 1984), 177–184. DOI: https://doi.org/10.1145/390010.808263

[48] David Padua and Jeremy Wolfe. 1986. Advanced Compiler Optimizations for Supercomputers. *Commun. ACM* 29 (12 1986), 1184–1201. DOI: https://doi.org/10.1145/7902.7904

[49] Lu Peng, Wentao Shi, Jian Zhang, and Samuel Irving. 2019. Exploiting Model-Level Parallelism in Recurrent Neural Network Accelerators. 241–248. DOI: https://doi.org/10.1109/MCSoC.2019.00042

[50] Mirco Ravanelli, Philemon Brakel, Maurizio Omologo, and Y. Bengio. 2018. Light Gated Recurrent Units for Speech Recognition. *IEEE Transactions on Emerging Topics in Computing* 2 (03 2018). DOI: https://doi.org/10.1109/TETCI.2017.2762739

[51] Tony Robinson and F. Failside. 1987. Static and Dynamic Error Propagation Networks with Application to Speech Coding. 632–641.

[52] H. Sak, Andrew Senior, and F. Beaufays. 2014. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. *Proceedings of the Annual Conference of the International Speech Communication Association, INTERSPEECH* (01 2014), 338–342.

[53] Hojjat Salehinejad, Sharan Sankar, Joseph Barfett, Errol Colak, and Shahrokh Valaee. 2017. Recent Advances in Recurrent Neural Networks. (12 2017).

[54] Michael Schuster. 1999. On Supervised Learning From Sequential Data With Applications For Speech Recognition. (04 1999).

[55] Ilya Sutskever, James Martens, and Geoffrey Hinton. 2011. Generating Text with Recurrent Neural Networks. *Proceedings of the 28th International Conference on Machine Learning (ICML-11)* (01 2011), 1017–1024.

[56] Yuan Tang, Ronghui You, Haibin Kan, Jesmin Jahan Tithi, Pramod Ganapathi, and Rezaul Chowdhury. 2015. Cache-Oblivious Wavefront: Improving Parallelism of Recursive Dynamic Programming Algorithms without Losing Cache-Efficiency. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP* 2015 (02 2015). DOI: https://doi.org/10.1145/2688500.2688514

[57] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc Le, Wolfgang Macherey, Maxim Krikun, and Yuan et al. Cao. 2016. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. (09 2016).

[58] Minjia Zhang, Samyam Rajbhandari, Wenhan Wang, and Yuxiong He. 2018. DeepCPU: Serving RNN-based Deep Learning Models 10x Faster.